SiliconGraphics
Computer Systems

Group Membership Services

A Bunch of Folks

Revision No. - 0.970
Date of Last Revision - May 15, 1998

1. Introduction to Group Communication Services

1.1 Scope

Group Communication Services (GCS) is a distributed service layered on top of the IRIX Cluster Membership Service (CMS). CMS provides the abstraction of a *cluster of nodes* (a collection of clustered computers). CMS presents a consistent view of *node membership* in the presence of node and network failures.[1] GCS, in contrast, provides the abstraction of *process groups* - collections of *processes* distributed across a cluster, cooperating to provide a distributed application service. GCS presents applications with:

- a consistent view of *group membership* in the presence of process failures and changing node membership.
- an atomic messaging service.

Distributed *applications* use GCS to be notified of the normal termination or abnormal failure of individual *application instances*[2] running on a cluster node. While distributed applications must still undertake the task of instance recovery and reintegration, GCS relieves applications from the task of monitoring instance existence and liveliness in a distributed environment. GCS also provides a reliable communication service to simplify instance recovery and reintegration.

1.2 Architecture

GCS is implemented as a collection of GCS *instances*, one instance being active on each node of the cluster. The terms GCS instance and Group Communication Daemon (GCD) are used interchangeably throughout this document. GCD itself is a distributed application which uses Cluster Membership Services to maintain a consistent view of node membership across all instances. An application or client process joins a GCS group by registering with the local GCS instance. It leaves the process group by unregistering with the local GCS instance. In both cases, all group processes have a consistent view of group membership.

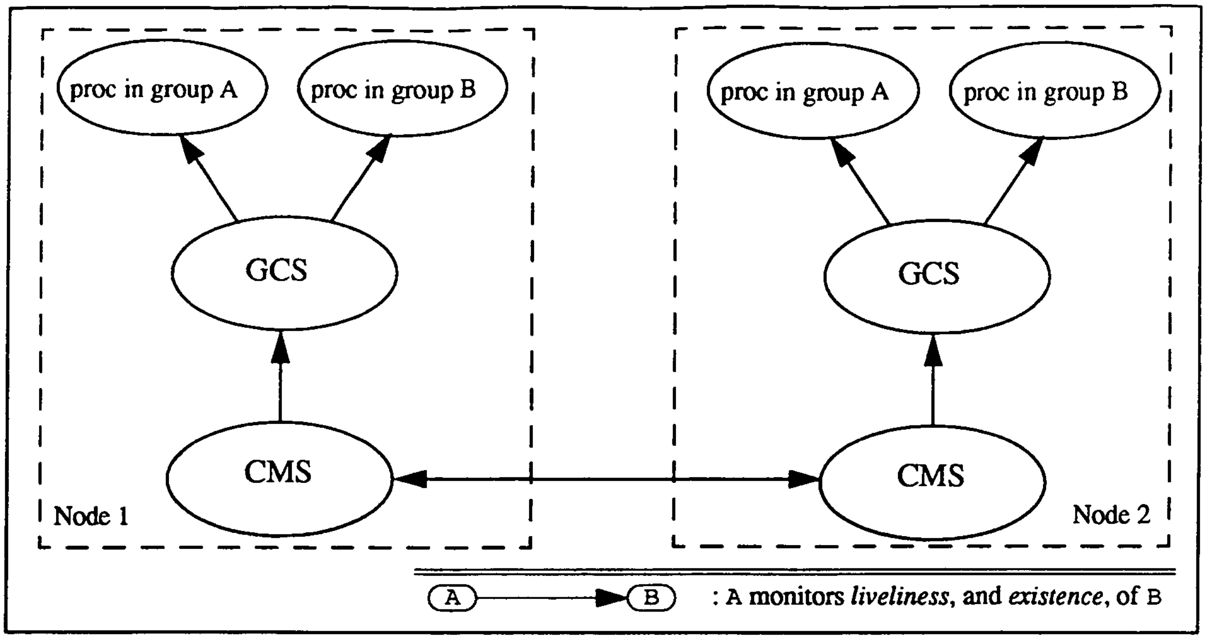
1.3 Failure Model

GCS operates in the context of a cluster as defined by CMS. If CMS excludes a node from the CMS membership group, GCS will exclude all processes running on the excluded node from their respective GCS groups. GCS is a *critical* client of CMS, i.e. if GCS is not running on a node, or fails, CMS will exclude the node from the CMS membership group.

GCD monitors the *existence* and *liveliness* of all processes within a group. Group process failures trigger a group membership change, with the failed process being reported as exiting in an *unknown* state.

1. For a more complete description of CMS, and its failure model, see *Cluster Membership Services*, described in the bibliography as Castellano[95].
2. In this document we use the terms *group member* and *application instance* interchangeably since they refer to the same entity - an application process registered with GCS at a node.

Figure 1: Process Architecture for SGI Distributed Services



1.4 Group Membership Protocol

The group membership protocol propagates group memberships across the cluster nodes using an ordered reliable broadcast. *Ordering* ensures that if two different GCS instances each try to propose a membership change the changes are processed in the same order at every GCS instance. *Reliability* ensures that the membership change is either accepted in its entirety, or not at all; *i.e.* if the GCS instance at any one node receives the message, GCS instances at all nodes are guaranteed to have received it.

1.5 What's Unique about GCS?

1.5.1 Atomic Messaging

GCS exposes an ordered, atomic messaging service to processes belonging to its groups. We base our definition of atomic upon that provided by Mullander[93]. This definition requires 4 properties to be upheld. Within our document, *reliable* is defined to mean a message satisfies the following three properties:

- *Validity*: if a correct process receives a message 'm', all correct processes receive 'm'.
- *Agreement*: if any correct process receives 'm', all correct processes receive 'm'.
- *Integrity*: for any message 'm', every correct process receives 'm' once, and only if it has been broadcast.

Informally, these three properties can be restated as: "all processes receive the message or none receive it and there are no spurious messages".

Also within this document, *atomic* is assumed to mean, in addition to satisfying the reliable property defined above, a message also satisfies the following property:

- *Atomicity*: if correct processes P and Q receive message 'm1' and 'm2', then P receives 'm1' before 'm2' if and only if Q receives 'm1' before 'm2'.

Informally, this property can be restated as: "all processes receive all messages in the same order".

1.5.2 Virtual Synchrony

In addition, GCS ensures that the same atomicity exists between messages and memberships. Another way to put this is each process in a group membership is presented with the same view of all participants in the group. This view may change over time but between two consecutive views, all participants in both views receive the same messages. Thus, every message is delivered within the view in which it was sent. This notion was first described by Birman[87] and is generally called *virtual synchrony.*

Informally, a total ordering is imposed upon all GCD messages, within a specific group membership, whether the messages originate from outside GCD (client messages) or whether the messages originate from inside GCD (membership changes).

1.5.3 Agreement on History

The Cluster Membership Services paper introduced the fundamental property of *agreement on history (aoh)*, originally defined by Cristian[88]. Group Membership Services also enforces that *agreement of history* property with respect to group memberships. A detailed discussion of *aoh* is provided in the Cluster Membership Services document, however the major implication of satisfying this property is that *each and every* group membership *must* be delivered to each correct process in the group, in the same order. Different processes in the group will never see different membership histories.

2. GCS Distributed Services

2.1 Process Architecture

GCS is implemented as a collection of GCD processes, one process being active on each node of the cluster. Each GCD process registers with the local CMS service when the GCD process starts. If CMS reports the node to be part of a cluster membership, the local GCD process completes its initialization. From this time forward, GCD allows application instances to register themselves as belonging to a process group.

2.2 Protocols

In what follows, the GCD process that initiates a protocol is called the *initiator.* The GCD process running on the oldest node in the cluster is called the *coordinator.* The identification of the oldest node in the cluster is provided by CMS. How CMS determines the oldest node in the cluster is discussed in the CMS design document. All GCD processes that are active during protocol execution are referred to as *participants.*

The *delta* protocol is initiated by a GCD instance to add/remove a local application instance from a particular group membership when the application instance registers/unregisters with GCD and to transmit an arbitrary message to other groups in the cluster. The membership changes actually form a subset of the general message set.

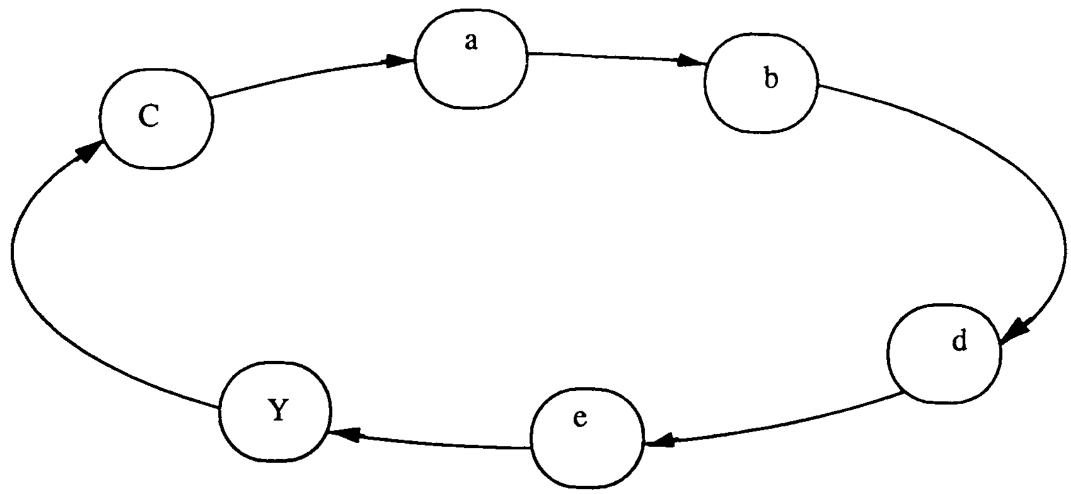
When a critical application instance exits without unregistering or fails to respond to monitoring in a timely fashion, the local GCD may exit, causing the local CMS instance to exit, which in turn causes the node the instances were running on to be physically reset. Exit or failure of a non-critical application instance is treated as an implicit unregister. The remaining GCDs are notified of the change in the group's membership.

When CMS detects a node crash (i.e. exclusion of a node from the cluster membership group), each participant re-evaluates the identity of the coordinator.

2.3 Instance Connectivity and Topology

Each participant opens a single connection to its next most younger sibling. Thus, all participants are connected in a ring, including the coordinator. It is an important characteristic of the GCS protocol definition that the ring is defined by the age of each node. In this way, rebuilding the ring in the presence of failures is fairly straightforward. Each node connects to its next younger sibling, except for the youngest node which connects to the coordinator. The coordinator is always the oldest node. Thus, the age of a node becomes a critical feature of the underlying CMS service.

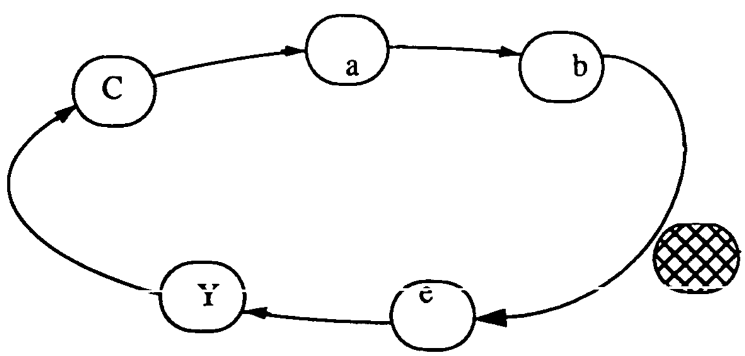
Figure 2: Connectivity of GCS instances



C: Coordinator (oldest node)
Y: Youngest node

If the coordinator crashes, as determined by CMS, each participant re-evaluates the identity of the coordinator. If the coordinator crashes the participants establish a connection with the new coordinator. If some other node crashes the ring is repaired by the preceding node connecting with the succeeding node, as shown in the figure below.

Figure 3: Ring repair when a node crashes



C: Coordinator (oldest node)
Y: Youngest node

2.4 GCD Incarnation Numbers

Each GCS instance or gcd has an incarnation number associated with its current life. This number is a monotonically increasing number, and is used to differentiate messages coming from different incarnations of a gcd on a given node. Each message originating from a node carries the incarnation number of that node's gcd. Incarnation numbers are used for several purposes, including:

- To ensure FIFO of messages originating at a node.
- To identify duplicate messages.
- To identify *old* messages, that can be safely thrown away.

These are described in detail in later sections of this document.

2.5 The Delta Protocol

The delta protocol is used by GCD to deliver ordered, atomic communications. These messages may be created by application instances and may contain arbitrary information, or they may be created by GCD itself and may be used to deliver membership information to other GCS instances in the cluster.

It is generally agreed that a two phase protocol for process group membership is required. We decided not to pursue the possibility of using a single phase protocol which did not guarantee the agreement on history property. A general purpose infrastructure should cater to a common denominator and if at least one major application requires a property, we felt the infrastructure should provide it. Our two phase protocol is similar to the classical two phase commit protocol used in distributed database algorithms with some revisions to simplify the protocol and reduce the complexity of recoverability. It can be summarized as follows. An initiator GCD node sends a message to the GCD coordinator node. The coordinator sends this message to the other GCD nodes in the cluster and waits for an acknowledgment from these nodes. This is the first phase or *proposal* phase. After receiving an acknowledgment, the coordinator sends out a *commit* message to the other GCD nodes and then waits for acknowledgment of this message. The commit phase is the second phase. All messages within a specific group are serialized through the coordinator, allowing a global ordering of messages, within a group. Unlike the traditional two-phase commit algorithm, an arbitrary node may not abort either a proposal or a commit.

In order to guard against several types of failures, each node maintains two data structures: a buffer to temporarily hold pending proposals and a buffer to temporarily hold commits. One proposal and one commit buffer is allocated for each group. In addition to these buffers, the coordinator node allocates a pending queue for each group. The protocol, along with the data structures and error cases we considered, are described in the following 6 steps. Steps 1, 2, and 3 constitute the first phase of the protocol. Steps 4, 5, and 6 constitute the second phase. In all of the following steps, we assume GCS instance failures and node failures to be synonymous and we further assume that determination of a new coordinator is asynchronous and automatic.

1. Step 1: A GCS instance (referred to as the initiator of the message) sends a message to the GCS coordinator. The initiator will resend the message if the initiator does not see a proposal for the message within some predefined period of time (see steps 2 and 3). The message is acked by the initiator to its client (the originator of the message) when the initiator sees a commit for the message (see steps 4 and 5). When the message is received by the coordinator, it puts the message in the pending queue of the coordinator. We consider these failures:

    a. Coordinator failure: The initiator will not see a proposal for the message and the message will be resent by the initiator after some prespecified time has elapsed (to the new coordinator).

    b. Initiator failure: If the message is in the coordinator's serializing proposal queue, the coordinator will not send out the message. The coordinator will filter out all the messages in the pending message queue from the initiator. If the message proposal has already been sent, the message will be delivered to all remaining GCD nodes. Only after all nodes have seen this message will new memberships (including the membership containing the failed initiator) be delivered.

c. Application instance failure: A new process group membership message is sent by the initiator to the coordinator. At this point the app instance fails. The membership is delivered to all GCD nodes after which the initiator will deliver a second membership showing the app instance's change in status.

d. Lost messages: The message may only be lost when moving from initiator to coordinator. This defaults to case a above.

2. Step 2: The coordinator sends a proposal for the message (ostensibly the message) to the remaining GCD nodes. The message is passed from one node to the next. The message proposal loops back to coordinator after being passed on by all the nodes. Each of the nodes puts the message proposal in its proposal buffer. There is a proposal time-out for each message. If the time-out value is reached, the coordinator resends the proposal message. Nodes who have already seen the proposal message continue to pass it along, as many times as necessary. We consider these failures:

   a. Coordinator failure: If the message has been received by the next node (the new coordinator), the message proposal will be sent around by the new coordinator. If the proposal message has not been received by the new coordinator, the message is sent by the initiator after the "message-resend" time-out. The message is put into the serializing proposal queue of the new coordinator.

   b. Other GCS instances fail: It does not matter. A message to indicate the change in the process group membership is queued by the coordinator in its serializing proposal queue. This newest membership will be delivered sometime later.

   c. A GCD joins: The new GCD will ignore all messages until it has a client. Until that time, it merely acts as a conduit, passing along all messages.

   When the initiator receives the proposal message, it can remove this message from its initiate queue. It does not need to initiate it ever again because this msg is in the coordinator's proposal queue. In the case of the coordinator dying, it will also be in the new coordinator's proposal queue; this follows from the fact that this msg is in this gcd's proposal queue, so it must be there in the proposal queues of all gcd's older than this gcd; one of these older gcd's or this gcd will become the new coordinator. If this gcd is the coordinator itself, then the case of the coordinator dying is a moot one.

   When any non-coordinator gcd receives a non-duplicate proposal message (viz. the next message in that group), it flushes out its commit queue. The commit queue had stored the previous message (if any) to handle potential resends. However, once it receives the next proposal message, it implies that all gcd's have seen the previous committed message, so there will not be a need to resend this previous committed message (and hence it can be removed from the commit queue). This ensures that, for a group, the proposal and commit queues are not both populated at the same time for any gcd (this condition already holds true for the coordinator).

3. Step 3: When the message loops back to the coordinator, the coordinator assumes that the message has been acknowledged by all other GCDs.

4. Step 4: When the ack for the message proposal is received by the coordinator (step 3), the coordinator sends the commit message. It then marks the proposal message as committed. The message is sent around the ring, from one node to the next younger. On

each GCD, before sending the message to the next GCD, the message is cached and then forwarded to the next node in the ring. Once the forwarding is complete the cached message is delivered to members of the process group (the clients). All the nodes keep track of the commit messages in their commit buffers. This is required because a node could, at some point, become the coordinator due to node failures. There can be only one commit message per process group. There is a commit time-out for each message. If the time-out value is reached, the coordinator resends the commit message. Nodes who have already seen the commit message pass the message to the next node as often as necessary. We consider these failures:

a. Coordinator failure: If the commit message has been received by the new coordinator, the commit message is resent by the new coordinator. If the commit message did not reach the new coordinator, the message is in the proposal buffer of the new coordinator node. The message is copied to the new coordinator's pending queue and the proposal message is resent.

b. Other GCS instances fail: If the GCS instance has not received the commit message or has received the message and not delivered to its clients, it does not matter. A message to indicate the change in the process group membership, due to the failing node is queued by the coordinator in its pending messages queue. This newest membership will be delivered sometime later. If the failing GCD had delivered the message to the clients and failed before passing the message to next GCD in the ring, the commit message will resent by the coordinator after the commit time-out.

c. A GCS instance joins: The node will ignore all the commit messages until there is a client for the GCS instance. Until that time, it merely acts as a conduit, passing along all messages.

d. All nodes who have received commit message fail: Since all nodes keep track of the proposal messages, the new coordinator resends the messages from its proposal queue.

5. Step 5: When the commit message loops back to the coordinator, the coordinator assumes that the message has been acknowledged. It is assumed that all the process group members have received the message from the GCS instances. The coordinator removes the message from the commit queue and continues to process the next message from its pending queue. The pending queue is not a serializing queue: rather, the coordinator preserves FIFO for all messages in a group that originate from the same node by determining the next message as follows. The coordinator maintains a 2-dimensional array of the sequence numbers of the last message committed for each group on each node, and a 1-dimensional array of incarnation numbers of each node. An element of the sequence number array is updated when the corresponding message loops back to the coordinator in the commit phase (i.e., beginning of Step 5). An element of the incarnation number array is updated when the node receives a message from another with the new incarnation number for the first time. The sequence number of a message determines its ordering, and is assigned by the source GCS instance. These sequence numbers start at 1 for each group on each node, and increase consecutively. To pick the next message to propose, the coordinator goes down the pending queue of that group, searching for the earliest message (both in terms of the incarnation and sequence numbers) from the

node whose message is at the head of the queue. If the sequence number of the earliest message in the pending queue for this group-node pair is not the next one in sequence when compared to the sequence number of its last committed message, then this message is not ready to be sent out. The next node in this group is then considered and so on until either a valid message is found, or there are no messages in this group that can be proposed yet.

6. Step 6: The clients of a GCD exchange heartbeats with it. If the process group membership has changed, the GCD acknowledges the heartbeat with the process group membership information.

Some other assumptions made by this protocol are described here.

- The protocol assumes that if the message reaches a particular GCD, it will be sent to the clients of that node.
- At any point, only one message per process group can be going through the protocol steps 2, 3, 4, and 5.
- A GCS instance has a commit buffer for each process group and a proposal buffer for each process group. These buffers hold at most one proposal or commit message.
- All GCS instances have to maintain a commit buffer and a proposal buffer for all process groups in the cluster (even if the GCS instance does not have any clients in the process group). It is necessary because any node could become the coördinator and therefore must be able to continue the protocol where it stopped.
- The coordinator has a separate serializing pending queue for each group, to keep all the messages initiated by other GCS instances.
- Should a node go down, on every other node CMS will notify GCS of the event. The GCS instance acting as coordinator will determine which clients of which groups have vanished and the coordinator will generate new membership messages for the vanished clients.

The protocol has the following time-outs:

1. A coordinator time-out for proposal messages which are sent but not acked.
2. A coordinator time-out for commit messages which are sent but not acked.
3. An initiator time-out for messages sent to the coordinator but which never show up in a proposal message from the coordinator.
4. A general GCD time-out for clients not supplying heartbeats within a specific period.
5. Two general GCD time-out for heartbeats with CMS.

2.6 Group Membership Initialization and Updates for GCS

In response to a membership change, the GCS instances or Group Communication Daemons (GCDs) send only the incremental information around (as opposed to the total membership message). In addition to this avoiding memory hogging problems in the GCD, it also always maintains correctness of membership information: i.e., a change in membership message is never outdated by the current membership information that a GCD has (this scenario can result if total membership messages are sent around). However, each GCD must always have the total

membership information since it may become the coordinator at any time; this is accomplished by it getting the total membership information from the coordinator when it initializes, and then updating it with incremental membership changes throughout. The coordinator starts with an initial total membership of "empty".

2.7 Initialization Protocol for a New GCD

The initialization protocol that a gcd has to follow in order to obtain the initial total membership information is as follows:

2.7.1 New GCD Initialization Steps

1. On startup, a new gcd sends a *GCSMSG_NEWGCD* message to the coordinator.
2. For each group, it sets a flag *newgroup[gindex]* to true.
3. When it gets a *GCSMSG_NEWGCD_REPLY* for a group from the coordinator, or a membership message for a group with group_seq_no of 1, it sets the corresponding *newgroup[gindex]* flag to false. The first case is straightforward, and the second case deals with groups that got formed *after* the coordinator responded to a *GCSMSG_NEWGCD* request message. If a client registers locally (membership message with a seqno of 1), it does not set the *newgroup* flag to false, but does build a group structure. Not setting the flag prevents it from reading any group related information for an existing group, until it gets the *GCSMSG_NEWGCD_REPLY* for this group. It may happen that a new gcd may get a *GCSMSG_NEWGCD_REPLY* after having seen a message with a group_msg_seqno of 1 for this group (say if this message was going around at the same time when this gcd sent its *GCSMSG_NEWGCD* request).

   If a client wants to register with this gcd, the gcd will only accept the registration request if :

   - the *newgroup[gindex]* flag is *false* for this group, **or**
   - the *GCSMSG_NEWGCD* message has reached the coordinator - the local gcd can be sure of this only when this message comes back to it in the *PROPOSE* state. This is necessary in case the *GCSMSG_NEWGCD* message gets lost in transit. Then the client registration request will proceed without the local gcd having the total membership information about that group.

   A new gcd does need to accept registration messages under the above conditions and not wait to get a *GCSMSG_NEWGCD_REPLY* from the coordinator, since that may not happen for a group of which the client in question is the first member.
4. When it gets a *GCSMSG_NEWGCD* message back from the coordinator, it updates its initiating queue when in proposal phase. When it gets this message in the commit phase, it just drops it. This is one place where a unidirectional (non 2phase) would help.
5. If it gets any other type of message for a group, it will only process it if its *newgroup* flag for that group is false. It will forward all messages to its next gcd, as always.

   NOTE: There is no msg in the initiating queue corresponding to the *GCSMSG_NEWGCD_REPLY* messages.

2.7.2 Coordinator Response to A New GCD Request

1. When the coordinator gets a *GCSMSG_NEWGCD* message from another gcd, then it queues the group membership of all the existing groups in their corresponding group pending/proposal queues. The message type for these messages is set to *GCSMSG_NEWGCD_REPLY.*
2. It also sends the original *GCSMSG_NEWGCD* message back to the sender gcd. It is necessary to have the original *GCSMSG_NEWGCD* message sent around in order to tell the new gcd that the coordinator has received the *GCSMSG_NEWGCD* request. This is necessary for cases when there will be no *GCSMSG_NEWGCD_REPLY* messages - i.e. when no groups exist (at startup).

The coordinator gcd itself does not go through the **New Gcd Initialization Steps**; it sets the flag *newgroup*[*gindex*] for all groups to *false* when it starts up.

Both the *GCSMSG_NEWGCD* and *GCSMSG_NEWGCD_REPLY* messages go through the two-phase commit process described earlier with the new gcd as the destination id (a gcd instance id is defined to be 0); other GCDs just act as forwarders of such messages. Until the new gcd receives and processes a *GCSMSG_NEWGCD_REPLY* sent to it by the coordinator for a group, it itself serves as a message forwarder for that group as far as the other messages going around in the system are concerned.

2.8 GCS-CMS Integration

GCS needs to obtain information about node membership of the cluster from the CMS layer. To achieve this, a gcd process needs to register with CMS at startup and then periodically ask CMS for node membership changes. After registering with CMS, gcd sets up several ways of receiving information from CMS periodically.

- It asks CMS, via the *cms_notify*() call, to give it asynchronous notification (using signal delivery) of a change in the node membership. It asks CMS to send it a SIGUSR2 for this purpose.
- It registers as a *critical client* of CMS, via the *cms_monitor*() call, asking CMS to check for its liveliness every *cms_mon_time*l. This results in CMS sending gcd a SIGUSR2 periodically.
- It sets up a periodic pulsing timer to do voluntary CMS pulsing at least once every *cms_pulse_time*.

Whenever a gcd receives a SIGUSR2, it polls the CMS daemon (*cmsd*) for new node membership information. If it does not get this signal from cmsd for *cms_pulse_time,* gcd polls CMS anyway. This is necessary for cases when the cmsd has been killed and could not send gcd any notification.

The node membership information from CMS contains the number of nodes in the cluster at a given time, and for each of the nodes, the id, states, the active ip address, status, mode, incarnation, and age.

2.9 Node Membership Changes

If a node that is part of the current cluster changes state from *CMS_NODE_UP* to *CMS_NODE_DOWN* or to *CMS_NODE_UNKNOWN*, then that is equivalent to all GCS clients on that node being excluded from the group membership. The coordinator gcd originates these messages for each node that goes down or to an unknown state. No special processing is needed for a node whose state has just changed to *CMS_NODE_UP*.

2.10 Dynamic Addition and Deletion of Nodes to the Cluster

2.11 Notable Design Decisions

CMS can deliver 3 types of membership notifications to GCD. There are:

- Node Joining (CI_GCSLIBERR_UP)
- Node going away (CI_GCSLIBERR_DOWN).
- Node unknown (CI_GCSLIBERR_UNKNOWN).

The first two membership modifications are relatively self-explanatory. The third occurs when CMS detects a node going away and when CMS is unable to force a reset of the node. It assumes nothing about the state of the node and returns a node unknown state to GCD.

GCD handles the unknown state in the following way. For the purposes of the GCD ring, the unknown state is treated as though the node entered the down state. That is, GCD attempts to rebuild the ring excluding the node in the unknown state. For the clients, GCD does not interpret the state and returns to the clients new memberships of the type unknown. In this way, clients may interpret such a change in any way they see fit.

2.12 Client message buffering in GCD

If the GCD has a message for a particular client in its initiating queue, it will not accept further messages from that client until this message has been moved to the proposal state. This is done so that a client sending large number of messages does not hog too much memory causing GCD to fail due to lack of memory. Client can continue sending messages until it runs out of IPC buffer space.

3. GCS Interface Definition

3.1 Interface discussions

GCS is an enabling technology which makes use of other enabling technologies. As such, GCS has three sets of interfaces, or portals of communication. One set is used by clients who wish to use the group membership functions or who wish to use reliable messaging. The second set is used when GCS communicates with CMS. The third set is used to provide administrative and monitoring functions.

3.2 The CMS interface

The CMS interface is completely described by the CMS design document. GCD is considered a client of CMS and exchanges *heartbeats* with CMS. This means that GCD makes a call to an CMS function periodically, to let CMS know GCD is alive. In response to this call, CMS returns information to GCD telling GCD if there has been a recent change in the cluster membership.

GCD is configured to act as a *critical* client of CMS. This means if GCD were to exit abruptly, CMS would eventually detect the fact and would remove the node from the cluster membership.

The time duration between heartbeats is a configured parameter. In addition, it may be updated as the system runs. This is simply accomplished by requiring GCD to tell CMS, at every heartbeat, how much time must pass without a heartbeat before CMS considers GCD to be dead. For example, GCD may tell CMS that the crucial time-out should occur 10 seconds hence. GCD is free to heartbeat as frequently as it wishes during the next 10 seconds, from one heartbeat to many, provided it performs at least one. Should GCD decide to increase the duration between heartbeats, it is a simple matter for GCD to inform CMS at the time of next heartbeat. Obviously the maximum duration between heartbeats will become a trade-off between response time (the time which elapses between the time when CMS receives a new cluster membership and the time when it delivers the change to GCD) and resource consumption (cpu usage). Because the code path for this heartbeat requires no context switches or kernel calls, it is thought the pathlength will be very small. A very small pathlength allows a fairly high frequency heartbeat, with a corresponding good response time, but without a significant cpu consumption penalty.

3.3 The Client Interface

The client interface is formed of those calls or services which clients use as part of their basic functionality. These calls or services are: *gcs_pulse*, *gcs_send*, *gcs_receive*, *gcs_register*, and *gcs_unregister*.

Processes register with GCD to join a group. Processes unregister from GCD to leave a group. GCD monitors the existence and liveliness of group processes on the local node. GCD does not monitor processes across nodes of the cluster. Group processes do not monitor the liveliness of other group processes.

After registering with GCD to join the group, processes may send or receive messages. Finally, processes are required to make the *gcs_pulse* call periodically. The pulse period may be altered during runtime and, like the CMS heartbeat, should strike a balance between responsiveness and excessive resource consumption.

The notion of *virtual synchrony* between membership changes and delivered messages is kept intact, within a process group. A total ordering is preserved. For example, if GCD receives a new membership for group G and then receives two messages, also for clients in group G, the clients will not see the messages until such time as they have received the new membership. After that time, the messages will be received by the clients in the order they were received by GCD.

A particular client may belong to more than one group. Joining each group requires a unique call to register within that group. Clients are not required to call *gcs_pulse* once for each group they belong to.

3.3.1 Signal Handling

All the GCS functions are not reentrant. If the function `gcs_*()` is called from a signal handler for a signal `sig`, the programmer should make sure `sig` is masked off throughout the call to `gcs_*()`.

3.3.2 Errors and Error Handling

All GCS calls follow the same error reporting and handling structure. A non-zero return value from a function is an error. The particular value defines the specific GCS error condition.

```
int gcs_get_errno (void)
```

If the GCD error was ultimately caused by an operating system error, GCD saves, in a special location, the errno value which the operating system set. This value is not overwritten until a new GCD call is made which produces a new failure. In this way, the most recent errno value associated with a GCD call may be retrieved at any time. Clients should not test for errors using this function unless they have first encountered a GCD error return value. If the error returned by GCD was not caused by an operating system error, this function will return an errno of 0.

The only error message returned by this function is:
`CI_GCSLIBERR_NOGCS`:GCD is not operating.

3.3.3 Data Structures Used

gcs_hdl_t

```
typedef struct cl_register {
                list_t              list;
                int32_t             group_id;
                ci_instid_t         inst_id;
                ipcclnt_hdl_t       comm_handle;
                int32_t             last_unix_error;
                gcs_membership_t    memp;
                gcs_membr_diff_t    membr_diff;
                int32_t             msg_seqno;
                int32_t             group_msg_seqno;
} cl_register_t;

typedef  struct cl_register* gcs_hdl_t;
```

This is gcs library handle returned to the client upon successful registration with GCD.

gcs_msg_t

```
typedef struct gcs_msg {
                    uint32_t          sender_instid;
                    msg_type_t          msg_type;
                    uint32_t            msg_size;
                    char*               msg_ptr;
} gcs_msg_t;
```

This structure is used to send and receive messages to and from GCD.

`sender_instid`: This is assigned by GCD and is the instance id of the originator of the message.

`msg_type`: This indicates the type of message. Currently, it can be one of GCSMSG_CLIENT, GCSMSG_ADMIN or GCSMSG_MEMBERSHIP. Clients of GCS library can send messages of type GCSMSG_CLIENT and GCSMSG_ADMIN only. GCD will send messages of type GCSMSG_MEMBERSHIP to its clients.

`msg_size`: Contains size of message to be sent or received.

`msg_ptr`: This is a pointer to a buffer containing the actual message to be sent or received.

3.3.4 Registration and Unregistration

Each group client is required to register with GCD. A client may belong to multiple groups. Clients may register and unregister at any time and such activities will have no adverse bearing upon the remaining group members. This allows, for instance, clients to unregister while GCD is shut down and replaced and then for these clients to re-register. By calling unregister, the calling process indicates to its membership group that it is terminating gracefully. The semantics of graceful termination are determined by the application, not GCD. A process that exits without calling `gcs_unregister()` is declared to have failed.

To avoid sensitive timing windows, the function internally notifies GCD of the unregistration but does not immediately return to the client process. Instead, it continues to perform the equivalent of providing a heartbeat until the registration is acknowledged by GCD. This prevents the awkward situation in which an unregistration call causes a timeout which brings down a node.

Registration and un-registration are performed by the following functions:

gcs_register

SYNOPSIS

```
ci_err_t gcs_register (gcs_hdl_t *gcsh, int grp_id, short *inst_id,
int msg_pool_size, gcs_membership_t *members, int sig, clock_t
init_hb)
```

DESCRIPTION

This blocking function registers the calling process as instance `inst_id` in group `grp_id` with GCS. `msgpool_size` decides the size of buffer allocated for sending and receiving messages from the GCS. Current membership information is returned in `members`. A successful call to `gcs_register()` returns `CI_SUCCESS` and `gcsh` points to a valid *registration handle*. The registration handle is used to identify the

group and resources associated with the client in future calls to GCS. If `*inst_id` is set to `NULL`, GCD will choose a specific inst_id and return it to the caller. If a value is supplied, and that value is unique among all clients on the same node, then GCD will use that value. GCD will encode the node id of the particular node into the instance id when it uses the instance id. This encoding will be done transparently to the client process. Clients can request asynchronous notification of new messages by specifying signal `sig` to be delivered to them. Clients must specify a value for the initial heartbeat value, *init_hb*, that is used for heartbeating until this value is changed by the `gcs_pulse()` function described below. Clients may use the default value GCS_INITIAL_HB_INTERVAL defined in **gcs.h** for this parameter.

The registration handle must be passed as an argument to all subsequent GCS library calls.

If the GCS server (GCD) is initializing after starting up when a gcs_register() call is made, then the gcs_register() call will block till the GCD initialization is complete, and GCD is ready to accept local clients.

RETURN VALUE

A successful call will return CI_SUCCESS and `inst_id` will contain instance id of the client that is unique on that node. If a system call fails, appropriate error code is returned.

CI_GCSLIBERR_NOPERM: Process doesn't have proper permission for GCD.

CI_GCSLIBERR_BADID: invalid argument, either grp_id, or inst_id is incorrect. Currently, also means that a client has already been registered with this `grp_id` and `inst_id`.

CI_GCSLIBERR_INVAL: Invalid *init_hb* value (<=0) specified.

FUTURE RETURN VALUES

CI_GCSLIBERR_DUPREQ: process with the same grp_id and inst_id has already registered.

CI_GCSLIBERR_SHUTDOWN: GCD is shutting down. Request denied.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_NOGCS: GCD is not operating.

CI_GCSLIBERR_NOTREADY: GCD is not yet ready to accept registration requests for this group.

gcs_unregister

SYNOPSIS

```
ci_err_t gcs_unregister(gcs_hdl_t gcsh)
```

DESCRIPTION

The function un-registers the calling process with GCD. Upon success the call returns CI_SUCCESS, otherwise it returns an appropriate error code. Once a process unregisters from a group, it may no longer receive messages destined for that group. If a process is not registered with any group, it may not receive any messages.

RETURN VALUE

CI_SUCCESS upon successful completion.

CI_GCSLIBERR_BADHANDL: caller process has not registered with GCS.

FUTURE RETURN VALUES

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_NOGCS: GCD is not operating.

3.3.5 Pulse

All client processes must call the pulse function periodically. This function informs GCD that the client still exists and is in control of its own destiny. This routine performs a pulse for all registrations the client has made.

gcs_pulse

SYNOPSIS

```
ci_err_t gcs_pulse(int heartbeat)
```

DESCRIPTION

The time-out period may be altered by adjusting the `heartbeat` input parameter. The time by which at least one call to `gcs_pulse()` must be made will be determined by adding the heartbeat value to an internal clock, maintained by GCD. From the moment the call returns, its new status is as was requested.

RETURN VALUE

The function returns CI_SUCCESS if the client and GCD connection is alive and it has not been forced out of membership by GCD. Note that `gcs_pulse()` performs pulse for all groups the calling process is a member of. If pulse for a particular client fails, `gcs_pulse()` will simply log an error message and return CI_SUCCESS.

CI_GCSLIBERR_BADPARM: invalid heartbeat value.

FUTURE RETURN VALUE

CI_GCSLIBERR_ SHUTDOWN: GCD is shutting down.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_REREG: GCD is available again. Reregister.

CI_GCSLIBERR_NOGCS: GCD is not operating.

3.3.6 Send and Receive

GCD provides the ability for clients to send and receive ordered, atomic messages. GCD makes no attempt to interpret the contents of the message. It treats the message like a package of bytes and sends it to a destination.

gcs_send

SYNOPSIS

```
ci_err_t    gcs_send(gcs_hdl_t    ghdl,    ci_instid_tdest_inst_id,
gcs_msg_t *msg)
```

The function allows a client to send message to one of: all clients in the sender's group or a specific client in the sender's group. The addressing is done as follows:

- If the `dest_inst_id` is set to GCS_BROADCAST (-1), the message is sent to all clients in the sender's group, including the sender.
- If the `dest_inst_id` valid, the message is sent to that client, assuming it resides within the sender's group.

`gcs_send()` checks if the GCD can accommodate given message size specified in `msg->msg_size`. Maximum size of a message is limited by buffer size of GCD memory pool, which currently is 64K.

RETURN VALUE

If the send is successful, the function returns CI_SUCCESS. Otherwise one of the following error codes is returned:

CI_GCSLIBERR_BADHANDL: The client is not registered with GCD.

CI_GCSLIBERR_BADID: invalid argument, inst_id is incorrect.

CI_GCSLIBERR_NOMEM: No memory left in GCD or message is larger than what is acceptable to GCD. Request denied.

CI_GCSLIBERR_INVAL: Incorrect message type. A client can send messages of type GCSMSG_CLIENT or GCSMSG_ADMIN only.

FUTURE RETURN VALUE

CI_GCSLIBERR_BADPARM: invalid msg (bad length or bad pointers within msg structure).

CI_GCSLIBERR_SHUTDOWN: GCD is shutting down.

CI_GCSLIBERR_NOTYET: GCD has not yet processed the last message sent. Try again.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_REREG: GCD is available again. Reregister.

CI_GCSLIBERR_NOGCS: GCD is not operating.

The function `gcs_send()` will copy the message to an internal transfer buffer and return immediately or, if the transfer buffer is full, it will return immediately with an error. In this way, the function will appear to be synchronous in nature.

gcs_recv

SYNOPSIS

```
int gcs_recv(gcs_hdl_t ghdl, gcs_msg_t *msg, boolean check)
```

DESCRIPTION

The function receives messages sent from other clients within its process group. The receive call is a non blocking call which means if no messages are waiting, the function returns right away. If no messages are waiting when receive is called, an

error code is returned. The parameter `check` can be used to specify that `gcs_recv()` was called to find out if there was a message pending. In this case, if `check` is set to TRUE and there is no message, CI_IPCERR_NOMSG is returned, but the the function does not log this event as an error. The purpose of this flag is to control log message. This function is also used to receive new group memberships. The structure returned in the message containing the new membership also contains information describing whether the membership change was an addition or a deletion from the group and, if it was a deletion, whether the deletion was graceful or not graceful. The function expects the size of message to be received be less than or equal to the size specified by `msg->msg_size`.

RETURN VALUE

If the receive is successful, the function returns CI_SUCCESS. An unsuccessful receive does not necessarily mean no message is waiting. Some early error codes are listed here:

CI_GCSLIBERR_BADHANDL: The client is not registered with GCD.

CI_GCSLIBERR_NOMSG: no message present.

CI_GCSLIBERR_NOSPACE: The buffer specified is not big enough to hold the new message.

CI_GCSLIBERR_BADPARM: invalid msg (bad pointers) or wrong flag for registration.

FUTURE RETURN VALUE

CI_GCSLIBERR_SHUTDOWN: GCD is shutting down.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_REREG: GCD is available again. Reregister.

CI_GCSLIBERR_NOGCS: GCD is not operating.

3.3.7 Membership Information

The following set of functions allow the client to obtain membership information. This information is limited to the current group of which the client is a member. If a client wishes to obtain information about other group, it must register with that group or belong to the administration group.

The following set of functions operate on the membership information contained in the gcs handle.

gcs_get_no_of_members

SYNOPSIS

```
uint32_t gcs_get_no_of_members(gcs_hdl_t gcsh)
```

DESCRIPTION

This function returns the number of client processes currently registered with this group.

gcs_get_instids

SYNOPSIS

```
ci_err_t   gcs_get_instids(gcs_hdl_t   gcsh,   uint32_t   *count,
ci_instid_t **inst_ids)
```

DESCRIPTION

This routine returns an array of global instance ids of clients registered within this group. The number of instances is returned in `count`. The caller of this routine is responsible for freeing memory pointed to by `inst_ids`. It sets `count` to the number of instances if `inst_ids` is set to NULL.

RETURN VALUE

CI_SUCCESS upon success.

CI_FAILURE: if count is NULL

CI_GCSLIBERR_NOMEM: if it is unable to malloc memory for inst_ids.

gcs_get_instids_on_node

SYNOPSIS

```
ci_err_t gcs_get_instids_on_node(gcs_hdl_t gcsh, ci_nodeid_t no-
deid, uint32_t count, ci_instid_t ** inst_ids)
```

DESCRIPTION

This routine is used to obtain an array of instance ids of clients running on a node whose id is given by `nodeid`. The number of instances found is returned in `count` . The caller of this routine is responsible for freeing memory allocated to `inst_ids`.

If `inst_ids` is set to NULL, the function simply returns the number of clients on that node.

RETURN VALUE

CI_SUCCESS upon success.

CI_FAILURE: if count is NULL

CI_GCSLIBERR_NOMEM: if it is unable to malloc memory for inst_ids.

gcs_get_member_nodes

SYNOPSIS

```
ci_err_t  gcs_get_member_nodes(gcs_hdl_t  gcsh,  uint32_t  *count,
ci_nodeid_t** nodeids)
```

DESCRIPTION

This routine is used to obtain an array of `nodeids` of nodes in the current membership. The number of nodes is returned in `count`. The caller is responsible for freeing memory pointed to by `nodeids`.

RETURN VALUE

CI_SUCCESS upon success.

CI_FAILURE: if count is NULL

CI_GCSLIBERR_NOMEM: if it is unable to malloc memory for inst_ids.

gcs_is_member

SYNOPSIS

```
boolean_t gcs_is_member(gcs_hdl_t gcsh, ci_instid_t inst)
```

DESCRIPTION

Returns B_TRUE if `inst_id` is a member of the the client's group and return B_FALSE otherwise.

gcs_has_member_on_node

SYNOPSIS

```
boolean_t gcs_has_member_on_node(gcs_hdl_t gcsh, ci_nodeid_t no-
deid)
```

DESCRIPTION

Returns B_TRUE if the client's group contains an instance on the node specified by `nodeid`. Returns B_FALSE otherwise.

The following set of functions operate on the membership structure `gcs_membership_t` .

gcs_get_membership

SYNOPSIS

```
gcs_membership_t*   gcs_get_membership_copy(gcs_hdl_t gcsh)
```

DESCRIPTION

It returns a pointer to a copy of the membership information contained in the client handle. This membership structure can be free() by calling gcs_destroy_membership().

RETURN VALUE

Pointer to gcs_membership_t structure in success else returns NULL.

gcs_mem_get_no_of_members

SYNOPSIS

```
uint32_t gcs_mem_get_no_of_members(gcs_membership_t *memp)
```

DESCRIPTION

This function returns the number of client processes currently registered with this group.

gcs_mem_get_instids

SYNOPSIS

```
ci_err_t   gcs_mem_get_instids(gcs_membership_t   *memp,   uint32_t
*count, ci_instid_t **inst_ids)
```

DESCRIPTION

This routine returns an array of global instance ids of clients registered within this group. The number of instances is returned in `count`. The caller of this routine is responsible for freeing memory pointed to by `inst_ids`. It sets `count` to the number of instances if `inst_ids` is set to NULL.

RETURN VALUE

CI_SUCCESS upon success.

CI_FAILURE: if count is NULL

CI_GCSLIBERR_NOMEM: if it is unable to malloc memory for inst_ids.

gcs_mem_get_instids_on_node

SYNOPSIS

```
ci_err_t    gcs_mem_get_instids_on_node(gcs_membership_t    *memp,
ci_nodeid_t nodeid, uint32_t count, ci_instid_t ** inst_ids)
```

DESCRIPTION

This routine is used to obtain an array of instance ids of clients running on a node whose id is given by `nodeid`. The number of instances found is returned in `count` . The caller of this routine is responsible for freeing memory allocated to `inst_ids`.

If `inst_ids` is set to NULL, the function simply returns the number of clients on that node.

RETURN VALUE

CI_SUCCESS upon success.

CI_FAILURE: if count is NULL

CI_GCSLIBERR_NOMEM: if it is unable to malloc memory for inst_ids.

gcs_mem_get_member_nodes

SYNOPSIS

```
ci_err_t gcs_mem_get_member_nodes(gcs_membership_t *memp, uint32_t
*count, ci_nodeid_t** nodeids)
```

DESCRIPTION

This routine is used to obtain an array of `nodeids` of nodes in the current membership. The number of nodes is returned in `count`. The caller is responsible for freeing memory pointed to by `nodeids`.

RETURN VALUE

CI_SUCCESS upon success.

CI_FAILURE: if count is NULL

CI_GCSLIBERR_NOMEM: if it is unable to malloc memory for inst_ids.

gcs_mem_is_member

SYNOPSIS

```
boolean_t  gcs_mem_is_member(gcs_membership_t  *memp,  ci_instid_t
inst)
```

DESCRIPTION

Returns B_TRUE if `inst_id` is a member of the the client's group and return B_FALSE otherwise.

gcs_mem_has_member_on_node

SYNOPSIS

```
boolean_t    gcs_mem_has_member_on_node(gcs_membership_t    *memp,
ci_nodeid_t nodeid)
```

DESCRIPTION

Returns B_TRUE if the client's group contains an instance on the node specified by `nodeid`. Returns B_FALSE otherwise.

gcs_mem_get_first_instance

SYNOPSIS

```
ci_instid_t gcs_mem_get_first_instance(gcs_membership_t *memp)
```

DESCRIPTION

Returns the first instance in the membership list. If the membership list is empty, it returns INVALD_INSTID.

gcs_mem_print_membership

SYNOPSIS

```
void gcs_mem_print_membership(gcs_membership_t *memp, int loglvl)
```

DESCRIPTION

This function can be used to log membership structure at the logging level specified by `loglvl`.

gcs_get_membership_diff

SYNOPSIS

```
gcs_membrship_diff_t* gcs_get_membership_diff(gcs_hdl_t gcsh)
```

DESCRIPTION

This returns a pointer to a structure containing the difference in previous and current membership. The contents of the location pointed to will change when clients receive new membership messages. The structure contains an array of global instance ids of the clients that caused the change and the actual cause. The cause can be any one of the following:

GCSCHG_INSTANCE_FAILURE: The client failed to send pulse at regular interval.

GCSCHG_INSTANCE_UNREGISTER: The client has unregistered.

GCSCHG_INSTANCE_JOIN: The client has joined this group.

GCSCHG_NODE_FAILURE: The node on which the client was running has gone down.

GCSCHG_NODE_UNKNOWN: The node on which the client was running has gone to an unknown state.

GCSCHG_UNKNOWN: The client has gone to an unknown state.

3.3.8 Miscellaneous

gcs_get_msg_seqno

SYNOPSIS

```
int32_t gcs_get_msg_seqno(gcs_hdl_t gcsh)
```

DESCRIPTION

Use this function to obtain message sequence number of the last message received by the client for the group indicated by `gcsh`. This is also the sequence number allocated by the local GCD when sending out the message.

gcs_get_group_msg_seqno

SYNOPSIS

```
int32_t gcs_get_group_msg_seqno(gcs_hdl_t gcsh)
```

DESCRIPTION

This gives the message sequence number of the last received message by the client for this group. This sequence number is allocated by the co-ordinator.

gcs_get_globalid

SYNOPSIS

```
uint32_t gcs_get_globalid(gcs_hdl_t gcsh)
```

DESCRIPTION

This returns the globally unique instance id of the client secified by `gcsh`. This is different than the

3.4 Shutdown and Node Control

gcs_shutdown

SYNOPSIS

```
int gcs_shutdown(gcs_hdl_t ghdl, gcs_flag_t flag)
```

DESCRIPTION

This command is used to cleanly shut down GCD on the local node. Once issued, GCD informs the clients it is shutting itself down. This is done by returning a specific error code to the client on the next GCS function call the client makes. GCD then proceeds to issue new memberships to the other GCS instances in the cluster, informing them of the pending membership changes (deletions) on its own node. After receiving all confirmations the GCS instance unregisters itself from CMS and exits. It does not wait for all clients to call *gcs_unregister()* but it does ensure the graceful membership deletions are made known to all other group members.

The `flag` variable provides the ability to shut GCD down with some special actions taken to allow upgrades to take place without bringing down applications. Two values are defined:

`GCS_STOP`: Stop GCD.

`GCS_FREEZE`: Stop GCD for upgrade.

The flag value `GCS_FREEZE` causes a special return code to be generated, informing clients that GCD will soon be back and that while GCD services are not available, this interruption is purely temporary. Clients may continue calling all functions they previously called, like *gcs_pulse*, *gcs_send*, and *gcs_recv* and they will continue to receive the GCS_FREEZE return code. Once the new version of GCD is back up, the GCS_FREEZE code will be replaced by the return code GCS_REREG. Upon seeing this return code, the client must re-register itself with GCD.

Even though GCS' intention is uninteruptible service, GCD must remove all registered clients from their group memberships before it stops. The GCS_FREEZE option is provided so that clients may be informed that the removal of GCD is a temporary event.

The flag value GCS_STOP causes the return code CI_GCSLIBERR_SHUTDOWN to be returned to all clients at the time of their next GCS call. Their memberships in their groups will be terminated. The clients will eventually be without GCS and must change their behaviors accordingly. After GCD completes its shutdown, further GCS calls will return CI_GCSLIBERR_NOGCS.

This function may only be invoked by clients with an effective userid equal to root.

3.5 Administration

Various sorts of applications have need for an interface explicitly separate from the membership and messaging interface. Such applications may include logging tasks, administrative tools, debugging tools, and performance monitoring tools. We define administrative interface which allows clients to obtain information about their group as well as other groups. Clients can obtain information about their group or change group specific parameters using this interface. However inorder to obtain information or change group specific parameters of other group, they must belong to a special admin group, the GCS_ADMIN_GROUP.

gcs_admin

SYNOPSIS

```
ci_err_t gcs_admin(gcs_hdl_t gcsh, gcs_admin_action_t act, void
*datap)
```

DESCRIPTION

This function allows clients to specify action *act* using information provided in *datap.* Action can be one of the following:

GCS_GROUP_INFO: Membership and timer values for group indicated by data

GCS_GLOBAL_SNAPSHOT:

GCS_LOCAL_STATS: Return information about number of clients and messages sent for this group.

GCS_SET_GROUP_PARAMS: Set group parameters like timer values, etc.

GCS_STIMULUS: Send test stimulus to GCD (used by GCD test suite only)

GCS_RESPONSE: Ask GCD to send response to an earlier stimulus (used by GCD test suite only).

RETURN VALUE

CI_SUCCESS: if the client can perform the operation successfully.

CI_GCSLIBERR_NOPERM: The client has requested an operation for a different group and it does not belong to GCS_ADMIN_GROUP.

CI_FAILURE: Data required for the requested operation was missing.

gcs_count_nodes (unimplemented)

SYNOPSIS

```
int gcs_count_nodes(gcs_flag_t flags)
```

DESCRIPTION

This function returns the number of nodes in the cluster. The *flags* variable may be used to specify query parameters. Three values are defined to limit the search. The possible values are:

CI_GCSLIBERR_UP: all nodes in the cluster currently up and running.

CI_GCSLIBERR_DOWN: all nodes in the cluster currently down and have been reset.

CI_GCSLIBERR_UNKNOWN: all nodes whose state is unknown.

If these three values are OR'ed together, *gcs_count_nodes()* returns the total number of nodes in the cluster.

RETURN VALUE

It returns CI_SUCCESS if no error occurs. Some early error codes are listed here:

CI_GCSLIBERR_NOPERM: process doesn't have proper permissions.

CI_GCSLIBERR_SHUTDOWN: GCD is shutting down.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_NOGCS: GCD is not operating.

gcs_coord_test (unimplemented)

SYNOPSIS

```
boolean_t gcs_coord_test(void)
```

DESCRIPTION

This function returns a boolean value depending upon whether the current GCS instance is the group coordinator.

RETURN VALUE

B_TRUE is returned if this GCS instance is the coordinator, B_FALSE if it is not.

If an error occurs, the function returns one of the following error codes. Some early error codes are listed here:

CI_GCSLIBERR_SHUTDOWN: GCD is shutting down.

CI_GCSLIBERR_NOGCS: GCD is not operating.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

gcs_count_groups (unimplemented)

SYNOPSIS

```
int gcs_count_groups(void)
```

DESCRIPTION

This function gives the number of groups with members on this particular node.

RETURN VALUE

It returns the count of such groups. Some early error codes are listed here:

CI_GCSLIBERR_SHUTDOWN: GCD is shutting down.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_NOGCS: GCD is not operating.

gcs_count_procs (unimplemented)

{i think this routine may not be required since gcs_admin() provides similar functionality.}

SYNOPSIS

```
int gcs_count_procs(int grp_id)
```

DESCRIPTION

This function returns the number of client processes registered with a given group.

RETURN VALUE

Some early error codes are listed here:

CI_GCSLIBERR_NOPERM: process doesn't have proper permissions.

CI_GCSLIBERR_BADPARM: flag is incorrect.

CI_GCSLIBERR_SHUTDOWN: GCD is shutting down.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_NOGCS: GCD is not operating.

gcs_get_mbrinfo (unimplemented)

SYNOPSIS

```
int gcs_get_mbrinfo(int grp_id, gcs_info_t *info,size_t size)
```

DESCRIPTION

This function retrieves the current membership information for the specified group into a buffer managed by GCD. It is important to note that this call may not be used for any protocol which requires the *Agreement on History* property to be satisfied. Such protocols must use the pulse/receive interface.

The function copies the membership at the time of the call to the area pointed by info. The size parameter specifies the size in bytes of the area pointed by info. There is one gcs_info_t structure per process in the group. Among other values, the structure includes the following members:

```
gcs_nodeid_t nodeid;/* node on which instance is active */
gcs_instid_t instid;/* instance within group */
gcs_status_t status;/* instance status */
```

status can be one of:

GCS_UP: instance is member of the group membership.

GCS_DOWN: instance is not a member of the group membership.

GCS_ERROR: instance has exited the group membership in an unknown state.

RETURN VALUE

If a membership is available the function returns CI_SUCCESS, otherwise it returns an error code indicating the problem. Some early error codes are:

CI_GCSLIBERR_NOPERM: calling process may not access this information.

CI_GCSLIBERR_BADPARM: size/info mismatch

CI_GCSLIBERR_SHUTDOWN: GCD is shutting down.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_NOGCS: GCD is not operating.

gcs_get_info() does not return group membership information in a serial order. For example, if more than one membership is generated in the real time interval between any two calls to **gcs_get_info()**, the call will return the most current membership.

gcs_get_counter (unimplemented)

SYNOPSIS

```
long gcs_get_counter(int grp_id, gcs_flag_t flag)
```

DESCRIPTION

GCD defines several counters which track several different types of events. All counters are 64 bit counters and all counters begin counting when GCD initializes. Counters may not be cleared. Any user may retrieve their value by calling this function and identifying the particular counter desired via the flags variable. The group id is used for identification. The addressing is done as follows:

- If the grp_id is set to -1, the counter value returned contains information specific to the entire node (GCS instance). Appropriate flags for this parameter setting are:
  a. MSGS_SENT: total number of messages sent by this GCS instance.
  b. MSGS_RECV: total number of messages received by this GCS instance.
  c. INIT_PROP: total number of initiator/coordinator time-outs.
  d. COORD_PROP: total number of coordinator proposal timeouts. This is only relevant if this GCS instance is the coordinator.
  e. COORD_COMM: total number of coordinator commit timeouts. This is only relevant if this GCS instance is the coordinator.
  f. MBRSHPS_ADD: total number of membership additions on this GCS instance, in all groups.
  g. MBRSHPS_DEL: total number of graceful membership deletions on this GCS instance, in all groups.
  h. MBRSHPS_CRASH: total number of non-graceful membership deletions (process crashes) on this GCS instance, in all groups.
- If the grp_id is valid, the counter contains information specific to the specified group. Appropriate flags for this parameter setting are:
  a. MSGS_SENT: total number of messages sent by this group.
  b. MSGS_RECV: total number of messages received by this group

c. MBRSHPS_ADD: total number of membership additions in this group.

d. MBRSHPS_DEL: total number of graceful membership deletions in this group.

e. MBRSHPS_CRASH: total number of non-graceful membership deletions (process crashes) in this group.

RETURN VALUE

The function returns one of the following error codes. Some early error codes are listed here:

CI_SUCCESS: Function call completed successfully.

CI_GCSLIBERR_BADID: grp_id variable is incorrect.

CI_GCSLIBERR_BADPARM: flag is incorrect.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_NOGCS: GCD is not operating.

gcs_get_config (unimplemented)

SYNOPSIS

```
long gcs_get_config(gcs_flag_t flags)
```

DESCRIPTION

GCS defines several configuration parameters which may be retrieved with these functions. The possible values are:

a. MAX_MESSAGE_SIZE: Size of max permitted message.

b. NUM_PROCESS_GROUPS: Number of statically defined process groups.

c. NUM_COORD_SER_QUEUES: Number of coordinator serialization queues. This number must either be one (1) or equal to the number of process groups.

d. SIZE_COORD_SER_QUEUE: Size, in bytes, of each coordinator serialization queue.

e. COORD_PROP_TIMEOUT: Time-out value used for coordinator proposal phase of the messaging protocol. The units are microseconds.

f. COORD_COMM_TIMEOUT: Time-out value used for coordinator commit phase of the messaging protocol. The units are microseconds.

g. INIT_PROP_TIMEOUT: Time-out value before initiator will resend to the coordinator. The units are microseconds.

h. CLIENT_HEARTBEAT_TIMEOUT: Maximum period before GCS instance assumes client will not supply a heartbeat and declares the client dead. The units are microseconds.

i. CMS_HEARTBEAT_TIMEOUT: Maximum period before CMS assumes GCS instance is dead. The units are microseconds.

RETURN VALUE

Some early error codes are listed here:

CI_SUCCESS: function call completed successfully.

CI_GCSLIBERR_BADPARM: flag is incorrect.

CI_GCSLIBERR_FREEZE: GCD is temporarily unavailable.

CI_GCSLIBERR_NOGCS: GCD is not operating.

3.6 Open Issues

The biggest open issue is that of security. For example:

- Is an open administrative interface a good thing? Security? e.g. anyone can get membership for any group?
- What about config files? Specifically, what about atomic updates? Do we need a **gcs_set_config()**? Do we need a **gcs_get_config()?**

3.7 Notable Design Changes

3.7.1 Removal of the "Critical" Process

The design originally specified a client process could register in a process group as either a *Critical* process or a *NonCritical* process. If a Critical process were to quit calling the GCS pulse routine, GCD would exit causing CMS to exit, causing the node to go down. This behavior was intended solely to provide what is known as *failstop* behavior. It is used when a critical application has failed in an indeterminate manner and it is of paramount importance that resources held by the application be released so they may be used or failed over to another application on another node.

We have changed the design so that the Critical process notion has been removed. The decision was made because GCD is not intended to assume any portion of the role of the FailSafe product. GCS provides both group membership services and reliable, ordered messaging in the presence of failures. It is not an enforcer of policy. Nodes within a cluster may still assume failstop behavior but the decision to reset will come from the FailSafe product and the reset will be generated by the FailSafe product and will be transparent to GCS. Should FailSafe itself fail, GCS will deliver this new membership to the remaining FailSafe processes and they may decide how to proceed.

4. GCS Internal Architecture

4.1 Logical Data structures

There are three primary data structures important in implementing the protocol defined herein. These three are

1. A per-node collection of proposal buffers.
2. A per-node collection of commit buffers.
3. The coordinator's serializing proposal queues.

All messages and all membership changes are globally ordered within a specific group membership. Memberships and messages are not guaranteed to be ordered between groups. On each GCS node, a buffer is allocated at GCD start time for each group, to contain a cache of the message proposal. Similarly, a buffer is allocated to contain the commit proposal. Thus, there are as many proposal buffers in a single instance of GCS as there are defined process groups. The same is also true for the commit buffers. These buffers are allocated at start time to ensure sufficient memory is available. As a result, the maximum number of groups supported by GCS is not a dynamically changing parameter. Rather it must be a configuration parameter.

The coordinator's serializing proposal queue contains many queues, one for each defined group. These queues are allocated when the GCS coordinator node starts. The queue is a predefined size and cannot grow dynamically so as to ensure sufficient memory is always available. The queue size is a configurable parameter.

4.2 GCD Memory Management

To ensure faster memory allocation and access, GCD uses memory management facility provided by the CHAOS library. All memory allocated by GCD comes from a pool of fixed size buffers that are pinned in the memory. The number of buffers allocated at any time depends of the memory used by GCD. Currently GCD uses buffers of 64K bytes and hence that is the maximum size for a single malloc request by GCD.

4.3 IPC mechanisms

Because of the simplicity and importance of the interactions between CMS and GCS, an effort has been made to ensure communications are reliable and not subject to failure during times of high system load. Indeed, it is during times of high system load that heartbeats become most important. For this reason, the interfaces are implemented without using any form of IPC which requires a trip through the operating system kernel.

Every effort will be made to use an IPC mechanism which doesn't use kernel services. There are three reasons for this:

1. Switching into the kernel means IPC calls are much more expensive. The cheaper our IPC calls are, the more of them we can make in a given time, increasing the responsiveness of the system without sacrificing resource consumption.
2. In environments where reliability and robustness are critical, using kernel IPC mechanisms means relying on code which may not be adequately tested and which may not be easily fixed, if problems are encountered.

3. Using kernel services allows for the potential exhaustion of kernel resources which may hamper IPC on heavily loaded machines. Kernel resources are typically beyond the control of user based programs and hence may not be adequately protected. One example of this scenario is mbuf exhaustion on heavily loaded machines which results in excessively lengthy tcp requests, causing false failure detections.

My current thinking is to use shared memory segments as communication buffers between GCS and its clients and between CMS and GCS. For all communications, protocols will be devised such that only reading and writing the shared segment will be required for communication. Such a design will have several assumptions:

- There exists one shared segment for each communication link.
- This shared segment is of a fixed size and does not grow or shrink.
- It is allocated by the GCD process at the time when a client registers.
- GCD will copy into the segment and the client will copy out. Implicitly, there will be 2 copies done for each IPC.
- The shared segment will be locked in memory, preventing paging.

4.3.1 Heartbeat mechanism design

A 64 bit virtual register is defined in the shared segment which contains the current time, to some arbitrary resolution. In addition, a variable is defined in the shared segment to hold the heartbeat value. The virtual time register and heartbeat cache are writable by the client process and readable by GCD. Whenever a client process calls **gcs_pulse()**, the heartbeat parameter is added atomically to the virtual time register and the heartbeat cache is atomically updated. In this way, the client can alter its heartbeat rate very easily. GCD can compare the current time against the virtual time register to determine whether the **gcs_pulse()** call is overdue. If so, it may use the heartbeat cache to determine just how overdue things really are.

Because there are no read/write structures shared between processes, no locks are needed.

4.3.2 Message transfer design

Three more data structures are defined in the shared segment. These structures are:

1. A 64 bit counting integer, named *old_value*, readable by GCD and writable by the client process.

2. A 64 bit counting integer, named *new_value*, readable by the client process and writable by GCD

3. A buffer containing the new message structure, writable by GCD and readable by the client process.

The protocol is then described by the following two protocols, each containing two steps. The client protocol is described first.

a. The variables *old_value* and *new_value* are compared. If they are equal, the client process returns as no new membership is available.

b. If *new_value* is greater than *old_value*, the client copies the membership structure from the shared segment to its own memory. It then atomically increments the variable *old_value*.

The GCD protocol is as follows:

a. The variables *old_value* and *new_value* are compared. If *new_value* is greater than *old_value* then we exit, to retry later, possibly during the next **gcs_pulse()** call.

b. If *new_value* is equal to *old_value*, GCD copies its new membership buffer to the shared segment. Once the copy is complete, it atomically increments *new_value*.

4.3.3 Message send/receive design

GCS uses the CHAOS IPC mechanism[1] for communication between its clients as well as with CMS. If a GCD client is not receiving messages fast enough, it may happen that GCD is unable to send more messages because its send buffer is full. In this case, the message is buffered in a queue for the group to which the client(s) belongs and GCD attempts to flush this queue periodically. If there are multiple clients belonging to the same group to whom GCD is unable to send message due to lack of buffer space, one copy of the message along with a reference count is stored.

4.3.4 Server Connection by the gcs_register function

The *gcs_register* function call presents a special case. After this call completes, every other call uses a handle returned by gcs_register to address the shared IPC mechanisms. The *gcs_register* function however does not have a handle nor any other mechanism with which to address the GCS demon. Some sort of mechanism must be provided with which clients may deliver shared memory identifiers to the server.

4.4 Versioning Support

We acknowledge that future versions of GCS and CMS may include changes which affect compatibility with the current design. We state here we will assume changes to the delta protocol will require stopping the operation of all CMS and GCS instances in the cluster. Other changes of similar magnitude will require stopping the operation of all CMS and GCS instances.

4.5 32 and 64 Bit Issues

The GCS product consists of a demon and a function call library. The demon will be a 32 bit entity, always. The function call library will be available in several versions: 32 bit and 64 bit. A client linked in 32 bit mode will use the 32 bit library while 64 bit clients will use the 64 bit library. The interface between the library call, executing in the context of the client, and the GCS demon will attempt to hide all "bitness" issue. We will use shared memory without pointers and explicit typing for data. We assume (although it doesn't really matter) that CMS will also be a 32 bit product.

4.6 Configuration parameters

Following is a list of all GCS configuration parameters and their default values.

- `MAX_MESSAGE_SIZE`: Default is 16 KB.
- `NUM_PROCESS_GROUPS`: Default is 16.
- `NUM_COORD_SER_QUEUES`: Default is `NUM_PROCESS_GROUPS`. One (1) queue may also be defined. This has the effect of serializing all message and all memberships with a corresponding decrease in performance.

1. For a complete description of CHAOS IPC refer to the *CHAOS Inter Process Communication* document.

- SIZE_COORD_SER_QUEUE: Default is MAX_MESSAGE_SIZE * NUM_PROCESS_GROUPS * Number of nodes in cluster.
- COORD_PROP_TIMEOUT: Default is 1 second.
- COORD_COMM_TIMEOUT: Default is COORD_PROP_TIMEOUT.
- INIT_PROP_TIMEOUT: Default is COORD_PROP_TIMEOUT * 2
- CLIENT_HEARTBEAT_TIMEOUT: Default is 5 seconds.
- CMS_HEARTBEAT_TIMEOUT: Default is 1/2 second.

Configuration parameters are initially defined by a configuration file. It is assumed this file exists on each node upon which GCS is expected to run and, further, that this file is identical upon each node. GCS will operate if the files are not identical but it will issue warning upon start-up. *How in the heck will we do this? And do we want to?*

5. Bibliography

1, Castellano[95]: *Array Membership Services,* Luca Castellano, Sharad Srivastava, 1995. Silicon Graphics Inc. proprietary design document.

2, Mullender[93]: *Distributed Systems*, Sape Mullender, 1993, Addison-Wesley.

3, Cristian[88]: *Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems,* Flavio Cristian, 1988, Springer Verlag.

4, Birman[87]: *Reliable Communication in the Presence of Failures*, Kenneth Birman and Thomas Joseph, 1987, ACM Trans on Computer Systems, vol 5, Number 1.

VERITAS®

VERITAS Volume Manager™
Administrator's Reference Guide
Release 3.0

Solaris
January, 1999
P/N 100-000917

VERITAS®

VERITAS Volume Manager™

Command Line Interface Administrator's Guide

Release 3.0

Solaris

February, 1999

P/N 100-000916

VERITAS®

VERITAS Volume Manager™

Getting Started Guide

Release 3.0

Solaris
January, 1999
P/N 100-000915